NMM Server XML RPC Interface

Motama GmbH, Saarbruecken, Germany (http://www.motama.com)

March 2011

Copyright (C) 2005-2010
Motama GmbH, Saarbruecken, Germany
http://www.motama.com

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being all sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found in the file COPYING.FDL.

THE DOCUMENT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE DOCUMENT BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENT OR THE USE OR OTHER DEALINGS IN THE DOCUMENT.

This document describes the XML-RPC interface of NMM server products. It can be used by custom applications, which are not necessarily using NMM, to communicate with the server product. It allows applications to control the server product or query information from the server product remotely using a HTTP connection.

1. Introduction

NMM server products, such as TVCaster (http://www.motama.com/tvcaster.html), RelayCaster (http://www.motama.com/relaycaster.html) and CodecCaster (http://www.motama.com/codeccaster.html), provide several interfaces for remote controlling its

operation or querying information. The web interface can be used for manual configuration of the server product by the user. The NMM interface allows applications using NMM to communicate with the server product using the RMI facilities of NMM. The XML-RPC interface is based on open standards, such as the HTTP protocol and the XML format, so it can be used by any application, such as a web application developed in PHP, a shell script, or a custom application which does not use NMM. Furthermore, our integrated RPC server supports encrypted connections using SSL.

2. XML-RPC Protocol

The XML-RPC protocol is a protocol for calling methods over the network. An RPC client connects to the RPC server and sends a request, which consists of a method name and a parameter list. The RPC server executes the requested method and sends a reply back to the client, which may contain a result value. Our integrated RPC server forwards all method calls to the server product software using the NMM interface and forwards replies from the server product back to the RPC client.

Requests and replies are represented as XML structures and embedded into HTTP requests and replies, which are transmitted using SSL over TCP/IP. The RPC server listens for incoming SSL connections at port 8111.

Here is an example of a request to a TVCaster product to create a stream in Corporate mode:

```
POST /RPC2 HTTP/1.0
User-Agent: PEAR XML_RPC
Host: tvcaster:8111
Authorization: Basic cnBjbm1tOnJwY2FybnVhbA==
Content-Type: text/xml
Content-Length: 697

<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
<methodName>rpc_createCorporateStream</methodName>
<params>
<param>
<value><string>192.168.1.73</string></value>
</param>
<param>
<value><int>0</int></value>
</param>
<param>
<value><int>1</int></value>
</param>
<param>
<value><string>192.168.1.31</string></value>
</param>
<param>
<value><int>12345</int></value>
</param>
```

```
<param>
<value><boolean>0</boolean></value>
</param>
<param>
<value><string></string></value>
</param>
<param>
<value><boolean>0</boolean></value>
</param>
<param>
<value><boolean>0</boolean></value>
</param>
<param>
<value><boolean>0</boolean></value>
</param>
<param>
<value><boolean>0</boolean></value>
</param>
</params>
</methodCall>
```

As you can see, requests are sent as HTTP POST requests. The request begins with the HTTP header, followed by an empty line, followed by the content as XML. The HTTP header contains the name of the host to which the request is being sent (`tvcaster`), the port number of the RPC server (8111) and the length of the content. The content contains the name of the method being called (`rpc_createCorporateStream`) and a list of arguments.

The corresponding reply from the server might look as follows:

```
HTTP/1.1 200 OK
Connection: Close
Content-Type: text/xml
Content-Length: 129
X-Powered-By: ulxmlrpcpp/1.7.4
Server: tvcaster
Date: Fri May  7 17:05:55 2010

<?xml version="1.0"
encoding="utf-8"?>
<methodResponse>
  <params>
    <param><value><i4>1</i4></value></param>
  </params>
</methodResponse>
```

The reply is sent as a standard HTTP reply, where "200 OK" means that the method call was successful. In this example, this means that the TVCaster has successfully created a stream. Similar to the request,

the reply consists of a HTTP header and XML content. The reply contains the return value "1", which in this example is the stream ID of the created stream.

3. Sending XML-RPC Requests

This section describes how to send XML-RPC requests to the server product and how to process responses. In general, this can be done using any XML-RPC library in any programming language. In this documentation, we describe how to use the "Ultra Lightweight XML-RPC library for C++" (ulxmlrpcpp). More information about this library can be found here:

http://ulxmlrpcpp.sourceforge.net/

The integrated RPC server of the server product can be reached under the following URL:

```
https://hostname:8111/RPC2
```

where "hostname" must be replaced by the host name of the product in the network, 8111 is the port on which the RPC server is running, and "/RPC2" is the resource name or directory name. Note that different XML-RPC libraries may provide different means of specifying these parameters.

The RPC server only supports secure (SSL) connections, and user/password authentification is required. Please see the user manual of your server product for information about setting the RPC password.

As a simple, but fully-featured example, we show how to create a stream on a TVCaster product in "Corporate" mode. The example creates a stream with fixed parameters, checks if the operation was successful and displays the numeric stream ID. Only stream creation is done using XML-RPC. The user must prepare the TVCaster product manually by setting a card to corporate mode and assigning a channel to it. Please see the TVCaster user manual and the comments in the example for more information. Some changes to the example are also required, such as the host name of the server product and the streaming destination, the RPC user name and the RPC password.

Here is the full source code of the example. It can also be found in the NMM-Server-SDK in the file `examples/tvcaster/xmlrpc_client.cpp`.

```
// Include headers of the XML-RPC library
#include <ulxmlrpcpp/ulxmlrpcpp.h>
#include <ulxmlrpcpp/ulxr_ssl_connection.h>
#include <ulxmlrpcpp/ulxr_http_protocol.h>
#include <ulxmlrpcpp/ulxr_requester.h>
#include <ulxmlrpcpp/ulxr_except.h>

#include <iostream>
```

```
using namespace std;
using namespace ulxr;

int main() {
  try {
    // Connection options.
    // You must edit the host name, user name and password depending on your
    // TVCaster configuration
    CppString host("192.168.1.74");
    unsigned int port = 8111;
    CppString resource("/RPC2");
    CppString user("rpcuser");
    CppString password("rpcpass");

    // Connection and initialization.
    SSLConnection conn(false, host, port);
    HttpProtocol prot(&conn);
    Requester client(&prot);

    // Call parameters for rpc_createCorporateStream.
    // This command creates a stream in corporate mode.
    // You must replace the destination host name and port by the actual
    // streaming destination. You may have to edit the card number and
    // channel number as well.
    // You must also prepare the TVCaster by configuring the given card in
    // "Corporate" mode and assigning a channel in the "Corporate Config" menu.
    // If the TVCaster is not properly configured, or a stream to the given
    // destination already exists, then the call will fail.
    // This example also demonstrates proper handling of failed requests.
    RpcString dvb_host(host);
    RpcString remux_host(host);
    Integer card(0);
    Integer channel(1);
    RpcString destination_host("192.168.1.31");
    Integer destination_port(12345);
    Boolean use_rtp(false);
    RpcString netsink_location;
    Boolean discard_sdt(true);
    Boolean discard_eit(true);
    Boolean generate_pat(true);
    Boolean descramble_service(true);

    // Create a method call with the arguments prepared above.
    MethodCall method_call("rpc_createCorporateStream");
    method_call.addParam(dvb_host);
    method_call.addParam(card);
    method_call.addParam(channel);
    method_call.addParam(destination_host);
    method_call.addParam(destination_port);
    method_call.addParam(use_rtp);
    method_call.addParam(netsink_location);
    method_call.addParam(discard_sdt);
    method_call.addParam(discard_eit);
```

```
    method_call.addParam(generate_pat);
    method_call.addParam(descramble_service);

    // Send call and get result.
    // User and password authentification is done here.
    MethodResponse response =
        client.call(method_call, resource, user, password);

    // Check if request was processed successfully and display result.
    if(response.isOK()) {
      // The operation was successful.
      // A stream has been created, and a stream ID was returned.
      // Display the stream ID and exit.
      const Integer& stream_id = response.getResult();

      cout << "SUCCESS! Stream ID: " << stream_id.getInteger() << endl;
      return 0;
    }
    else {
      // The operation has failed.
      // Error information has been returned. Display it and exit.
      const Struct& result = response.getResult();
      const RpcString& fault_string = result.getMember("faultString");

      cout << "FAILURE! " << fault_string.getString() << endl;
      return 1;
    }
  }
  catch(const Exception& e) {
    // The XML-RPC library has reported an error.
    cerr << "XML-RPC Error: " << e.why() << endl;
    return 1;
  }
  catch(const exception& e) {
    // Some other error has occurred.
    cerr << e.what() << endl;
    return 1;
  }
  catch(...) {
    // Some other error has occurred.
    cerr << "Unknown exception" << endl;
    return 1;
  }
}
```

4. XML-RPC API Reference

Full documentation of all available XML-RPC API functions for different products can be found in the following locations:

- TVCaster XML-RPC API (http://www.motama.com/doxygen-nmm-server-sdk-2.2.2/html/group__group__tvcaster__xmlrpc.html)
- RelayCaster XML-RPC API (http://www.motama.com/doxygen-nmm-server-sdk-2.2.2/html/group__group__relaycaster__xmlrpc.html)
- CodecCaster XML-RPC API (http://www.motama.com/doxygen-nmm-server-sdk-2.2.2/html/group__group__codeccaster__xmlrpc.html)